Shared Memory Access (SHMEM) Routines

Karl Feind, Cray Research, Inc., Eagan, Minnesota

ABSTRACT: *This paper gives an overview of the shared memory access (SHMEM) routines in the Libsma library on CRAY T3D™ systems. Routines are available which perform synchronization, data cache tuning, remote memory read and write, and collective operations such as broadcasting and reduction. Specific program examples are provided to help illustrate the usefulness and usability of the SHMEM routines.*

1 Introduction

The Libsma library contains an assortment of efficient routines which take advantage of the logically shared memory on CRAY T3D™ systems. A logically shared memory is one which allows any processor element (PE) to access memory in a remote PE without involving the microprocessor on the remote PE.

The Libsma shared memory (SHMEM) routines, together with Cray Research MPP Programing Model (CRAFT) and message-passing libraries such as PVM, offer the user several options for accomplishing some of the same tasks. The intent of this paper is to point out situations where use of SHMEM routines might be the right choice. It is also important to keep in mind that whether to use SHMEM, PVM, or CRAFT is not an "either-or" decision. The same program might use PVM for portability or CRAFT for simplicity, but use SHMEM in the bottleneck kernel of the application to enhance performance.

2 Alternatives to SHMEM: PVM and CRAFT

The PVM library offers a portable message passing paradigm. Unlike SHMEM, PVM is implemented on a wide variety of machine types, including cluster MPP systems, workstations, and logically shared memory MPP systems. SHMEM routines are designed with the assumption of logically shared memory, and to date have been implemented only on CRAY T3D™ systems.

The main disadvantage of the highly portable PVM interface (and any traditional message-passing interface) is that these interfaces have higher latency than for comparable SHMEM transfers. This extra latency is due to the extra functionality inherent in message-passing--especially support for buffering, queueing, and implicit synchronization--that is not needed by many MPP programs. In addition, having to match send and receive calls introduces unnecessary complexity to many programs.

A second alternative to using SHMEM routines is CRAFT. This programming model offers user-friendly data distribution and work sharing constructs. With user-friendliness, however, comes the disadvantage that the easily programmed code might not always map to an optimal pattern of global data access.

3 Top Ten Reasons to Use SHMEM

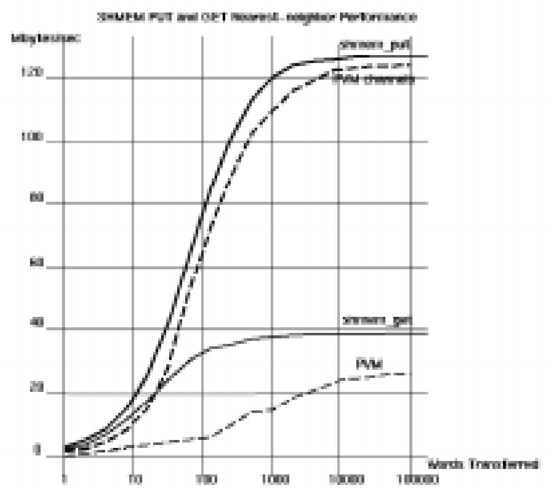
There are reasons to use SHMEM routines based on performance and functionality. The basic communication primitives (`shmem_get`, `shmem_put`, and `shmem_swap`) have very low latency and high bandwidth. Some SHMEM collective routines (for example `shmem_broadcast`), offer functionality not available directly via PVM. This section proposes that the top ten reasons to use SHMEM routines are:

1. Efficient remote data transfer.
2. Ease of use.
3. Efficient gather/scatter of remote data.
4. All-to-all data collections.
5. One-to-all broadcasting.
6. Atomic swap.
7. An assortment of reductions.
8. Locking of critical regions.
9. Barriers for task teams.
10. Other task team operations.

3.1 Efficient Remote Data Transfer

Perhaps the most familiar SHMEM routines are `shmem_put` and `shmem_get`. These routines transfer 64 bit words from a source to a target address. Copying from a remote source (on a different PE) to a local target is called a *GET* operation. The reverse is called a *PUT* operation.

PUTs have very low latency and very high bandwidth compared to the other methods for remote data transfer, including PVM channels, PVM, and GETs. The graph below compares transfer rates for the various remote data transfer methods.



The 64 bit `shmem_put` routine has a peak transfer rate of about 130 MBytes/sec., with half the peak rate achieved at transfer sizes of 65 words. The `shmem_get` routine has a peak transfer rate of about 40 Mbytes/sec., with half the peak rate achieved at transfer sizes of 14 words. Because GETs are much slower than PUTs, best performance is achieved if algorithms use PUTs.

Special-purpose PUT and GET routines are also available which transfer 32 bit data or strided data.

PUTs and GETs differ in their effect on the data cache. PUTs can cause the data cache on the receiving PE to become incoherent on CRAY T3D systems. Cache management routines such as `shmem_udcflush` and `shmem_set_cache_invalidate` must be used on the receiving PE in conjunction with PUT routines. GET routines, on the other hand, never make the cache invalid because memory update is done via memory stores which write through the local data cache.

Here is an example which puts 10 words from the `source` array on PE 1 to the `target` array on PE 2:

```
/* This program requires at least 3 PEs */
#include <mpp/shmem.h>
main()
{
    long source[10] = { 1, 2, 3, 4, 5,
                            6, 7, 8, 9, 10 };
    static long target[10];/*must be static*/
    if (_my_pe() == 1) {
      shmem_put(target, source, 10, 2);
    }
    barrier();      /* wait for completion */
    if (_my_pe() == 2) {
      shmem_udcflush(target);
      printf("target[0] on PE 2: %d\n",
          target[0]);
    }

}
```

3.2 *Ease of Use*

SHMEM interfaces can sometimes be simpler than more portable message passing paradigms. This is because management of tasks on clusters, for example, is typically more complicated than managing tasks for PEs on single machines. In addition, the one-sided style of communication which SHMEM offers (and which is made possible by an underlying logically shared memory system) is simpler because a matching receive request need not match every send request.

There are some complexities associated with SHMEM routines, for example the `pSync` and `pWrk` arrays which are illustrated in section 3.8. But for many cases, coding with SHMEM can be simpler and less error-prone than with PVM.

To get a taste of the difference between programming with SHMEM and PVM, the following PVM program is presented which accomplishes the same task as the previous program example which used `shmem_put`.

```
/* This program require at least 3 PEs */
#include <mpp/pvm3.h>
main()
{
    long source[10] = { 1, 2, 3, 4, 5, 6,
                      7, 8, 9, 10 };
    long target[10];
    int msgtag = 99999;
    if (_my_pe() == 1) {
      (void) pvm_psend(2, msgtag,
            (char*)source, 10,PVM_LONG);
    }
    if (_my_pe() == 2) {
      (void) pvm_precv(1, msgtag,
                    (char*)target, 10,PVM_LONG,
                    NULL, NULL, NULL);
      printf("target[0] on PE 2: %d\n",
          target[0]);
    }
}
```

The PVM program needs the receiver-side call to `pvm_precv` which matches the sender-side call to `pvm_psend`. In contrast, the SHMEM program needs to invalidate the cache associated with the receiving variable on the receiving side.

3.3 *Efficient Gather/Scatter of Remote Data*

One task which is simple and efficient to do with SHMEM is gather/scatter of remote data. This capability is not available with PVM (the `pvm_gather` and `pvm_scatter` functions are *not* indexed gather/scatter functions). The following example copies `SOURCE` array on PE 1 to the `TARGET` array on PE 0 with the order of the elements reversed:

```
!   This program requires at least 2 PEs

    ! SHMEM_IXGET requires that SOURCE have
    ! the SAVE attribute
    SAVE SOURCE
    INTEGER SOURCE(4)
    INTEGER TARGET(4), SRCINDEX(4)
    INTRINSIC MY_PE

    IF (MY_PE().EQ.1) THEN
      DO I = 1,4
      SOURCE(I) = I
      ENDDO
    ENDIF

    ! Ensure that SOURCE is initialized
    CALL BARRIER()

    IF (MY_PE().EQ.0) THEN
      ! The gather will invert the order of
      ! SOURCE.  Note that the index array
      ! contains 0-based indices.
      DO I = 1,4
        SRCINDEX(I) = 4-I
      ENDDO

      CALL SHMEM_IXGET(TARGET, SOURCE,
     &     SRCINDEX, 4, 1)
      PRINT*,'PE ',MY_PE(),' TARGET=',TARGET
    ENDIF
    END
```

3.4 *All-to-all Data Collections*

The `shmem_fcollect` routine efficiently updates a target array on all PEs such that element `I` of the source array on PE `P` is copied to element `I*P` of the target array. This is similar to the PVM function `pvm_gather`, with the difference that the PVM function updates the target on only one of the PEs. A call to `pvm_gather` followed by a call to `pvm_broadcast` would produce the same result as one call to `shmem_fcollect`, but the call to `shmem_fcollect` is faster.

The following example C program uses `shmem_fcollect` to copy the value of variable `myvalue` from all PEs to the local `allvals` array:

```
#include <mpp/shmem.h>
#define NPE 4               /* assume 4 PEs */
main()
{

    static long myvalue;
    static long allvals[NPE];
    static long
       pSync[_SHMEM_COLLECT_SYNC_SIZE];
    int i;

    for(i=0;i<_SHMEM_COLLECT_SYNC_SIZE;i++)
      pSync[i] = _SHMEM_SYNC_VALUE;
     myvalue = _my_pe();
    /*
     * Wait for pSync and myvalue to be
     * initialized on all PEs.
     */
    barrier();
    shmem_follect(allvals, &myvalue, 1, 0,
           0, _num_pes(), pSync);

    printf(
    "PE %d: collected values = %d %d %d %d\n",
              _my_pe(), allvals[0], allvals[1],
            allvals[2], allvals[3]);
}
```

The output is:

```
  PE 1: collected values = 0 1 2 3
  PE 3: collected values = 0 1 2 3
  PE 0: collected values = 0 1 2 3
  PE 2: collected values = 0 1 2 3
```

The pattern of remote memory transfer for `shmem_fcollect` is the same as the following CRAFT example:

```
        INTEGER SH(0:N$PES-1)
CDIR$ SHARED SH(:BLOCK)
        INTEGER PR(N$PES)
        INTRINSIC MY_PE

        SH(MY_PE()) = MY_PE()
        CALL BARRIER
        PR = SH
        PRINT*,'PE ',MY_PE(),' PR = ',PR
        END
```

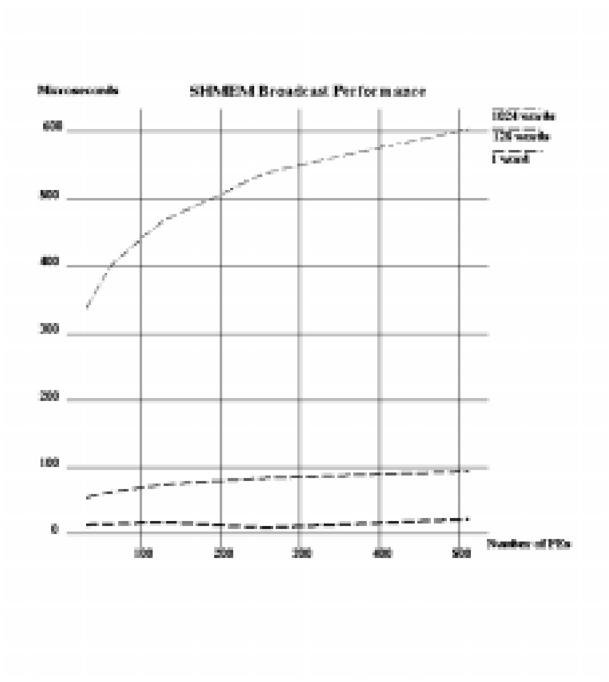
The output is:

```
  PE 0 PR = 0,  1,  2,  3
  PE 2 PR = 0,  1,  2,  3
  PE 1 PR = 0,  1,  2,  3
  PE 3 PR = 0,  1,  2,  3
```

3.5 *One-to-all Broadcasting*

The `shmem_broadcast` routine copies an array on one PE to the target array on all other PEs. The broadcast algorithm organizes the PEs into a binary tree and broadcasts the data by fan-out through the tree.

Below is a graph which shows the logarithmic performance of `shmem_broadcast` which arises from the efficient fan-out algorithm.



Here is a simple Fortran program which calls `SHMEM_BROADCAST`:

```
        INCLUDE “mpp/shmem.fh”
        INTEGER PSYNC(SHMEM_BCAST_SYNC_SIZE)
        DATA PSYNC
    & /SHMEM_BCAST_SYNC_SIZE*SHMEM_SYNC_VALUE/
        INTEGER TARGET(10), SOURCE(10)
        SAVE TARGET, SOURCE
        CALL SHMEM_BROADCAST(TARGET, SOURCE,
    &   10, 0, 0, 0, N$PES, PSYNC)
        END
```

The `shmem_broadcast` routine is much faster than `pvm_broadcast`, which is the PVM routine that provides the same capability.

3.6 *Atomic Swap*

The CRAY T3D hardware provides support for atomic swap operations on remote or local memory. An atomic swap operations reads the current value of a memory word and updates the same word as one atomic operation. The SHMEM routine `shmem_swap` is an interface to this hardware capability.

This short Fortran program demonstrates how to call `SHMEM_SWAP`:

```
        INTEGER SHMEM_SWAP
        INTEGER TARGET, NEWVALUE, OLDVALUE
        SAVE TARGET

        TARGET = 33
        NEWVALUE = 44
        OLDVALUE = SHMEM_SWAP(TARGET, NEWVALUE, 0)
        PRINT*,'TARGET=',TARGET
        PRINT*,'OLDVALUE=',OLDVALUE
        PRINT*,'NEWVALUE=',NEWVALUE
        END
```

When run on one PE, the following output results:

```
TARGET=44
OLDVALUE=33
NEWVALUE=44
```

3.7 *An Assortment of Reductions*

A family of SHMEM reduction routines are available for most common arithmetic functions and most data types. Supported operations include:

- min
- max
- product
- sum
- or
- xor
- and

A reduction programming example may be seen in section 3.8. PVM provides most of the same reductions which are available and much faster with SHMEM.

3.8 *Locking of Critical Regions*

The `shmem_set_lock` and `shmem_clear_lock` functions use shared memory to implement mutual exclusion locking. CRAFT programmers have access to the similar `SET_LOCK` and `CLEAR_LOCK` functions, but C programmers cannot use these functions because they require use of CRAFT distributed arrays. The following example program uses `shmem_set_lock` and `shmem_clear_lock` to protect a critical region and the `shmem_int_min_to_all` reduction function to detect which PE traversed the critical region in minimum time.

```
#include <mpp/shmem.h>
main()
{
    static int
      pWrk[_SHMEM_REDUCE_MIN_WRKDATA_SIZE];
    static long
      pSync[_SHMEM_REDUCE_SYNC_SIZE];
    static int mintime;
    static int mytime;
    static long slock;
    int t0, t1, i;
    char *chp;

    for (i=0; i<_SHMEM_REDUCE_SYNC_SIZE; i++)
      pSync[i] = _SHMEM_SYNC_VALUE;

    barrier();

    t0 = _rtc();
    shmem_set_lock(&slock);
    do_work();      /* critical region */
    shmem_clear_lock(&slock);
    t1 = _rtc();

    mytime = t1 - t0;

    /* find the PE with minimum time */
    shmem_int_min_to_all(&mintime, &mytime,
        1, 0, 0, _num_pes(), pWrk, pSync);

    chp = mintime == mytime ? “<---” : “”;
    printf(“PE %d mytime=%d mintime=%d %s\n”,
        _my_pe(), mytime, mintime, chp);
 }

do_work()
{
    static int accum;
    accum++;
}
```

This yields the following output when run with 4 PEs:

```
PE 3 mytime=3074 mintime=3074 <---
PE 2 mytime=3107 mintime=3074
PE 1 mytime=3086 mintime=3074
PE 0 mytime=3158 mintime=3074
```

3.9 Barriers for Task Teams

Sometimes it is useful to partition the available PEs into groups which communicate among themselves. To synchronize the communication, a barrier call is needed which targets a specific subset of PEs. This is the function provided by the `shmem_barrier` routine.

As an example, the following program groups the PEs into pairs. Each PE sends the address of a stack variable to its partner so that PUTs and GETs could later be done using that remote address.

```
#include <mpp/shmem.h>

long pSync[_SHMEM_BARRIER_SYNC_SIZE];

main()
{
    int mainvar;
    int i;

    for (i=0; i<_SHMEM_BARRIER_SYNC_SIZE; i++)
      pSync[i] = _SHMEM_SYNC_VALUE;

    barrier();

    if (_my_pe() & 1)
      extra();
    else
      doit(&mainvar);
}

extra()
{
    int stackvar;
    doit(&stackvar);
}

doit(int *stv)
{
    static long in_box;
    static long out_box;
    int partner =  (_my_pe() ^ 1);  /* xor */
    int PE_start = (_my_pe() & ~1);

    out_box = (long)stv;
    shmem_put(&in_box, &out_box, 1, partner);
    shmem_barrier(PE_start, 0, 2, pSync);
    shmem_udflush_line(&in_box);
    printf(“PE %d partner’s stv addr: %x\n”,
      _my_pe(), in_box);
}
```

The output when run with 4 PEs is:

```
PE 3 partner’s stv addr: 60ffffeef0
PE 2 partner’s stv addr: 60ffffee20
PE 0 partner’s stv addr: 60ffffee20
PE 1 partner’s stv addr: 60ffffeef0
```

3.10 Other Task Team Operations

The example program in section 3.9 illustrated the use of task teams. In that case the teams consisted of 2 PEs each. SHMEM routines in general support task teams of any size, with optional striding between the PEs in a task team, or *active PE set*. The stride between PEs in an active PE set must be a power of 2.

The following SHMEM routines allow the specification of an active PE set:

- shmem_barrier
- shmem_broadcast
- shmem_collect

- shmem_fcollect
- the SHMEM reduction routines

PVM supports a more general form of task teaming with its *task groups*. Setting up PVM task groups is more complex and using PVM task groups can be much slower than SHMEM active PE sets.

4 Where to Get More Information

With the CrayLibs_M 1.2 release, two manuals are available which document SHMEM routines for C and Fortran users:

- SN-2516, *The SHMEM Technical Note for FORTRAN*
- SN-2517, *The SHMEM Technical Note for C*

In CrayLibs_M 1.2.0.3 and later, a full set of SHMEM man pages are available on-line. These man pages will be included in SR-2165, *The Application Programmer's Library Reference Manual* in the next major CrayLibs_M release. The `intro_shmem`(3) man page is a highly recommended source of SHMEM information for novice users.

Cray Research offers training class TR-T3DAPPL, "Cray T3D Applications Programming". This class gives an overview of CRAFT, PVM, and SHMEM programming on CRAY T3D systems.

5 Conclusion

The SHMEM routines in library Libsma offer a wide variety of subroutines to help accomplish tasks related to inter-processor communication on CRAY T3D systems. The advantages in areas of performance, usability, and functionality make the SHMEM routines a very viable alternative to portable message passing libraries and CRAFT.